**CO503: Practical 2 – Processor Customization**

**E/17/154**

**05/03/2022**

The practical uses FPGA design tools to create System-on-Chip with a customized NiosII Processor. The customization is done in order to improve the performance of modulo-2 division operation in Cyclic-Redundancy-Check (CRC) algorithm used in network devices.

**Tasks:**

1. Add a custom instruction to the existing MIPS ISA of the NiosII processor, for performing the modulo-2 division.
2. Implement the required hardware functionality to support the custom instruction using XOR and shift operations.
3. Use the newly added custom instruction in the CRC algorithm and compare its performance against pure software implementations.

**Implementation**

The hardware implementation was done using IP blocks with minimal component to support the CRC custom instruction.

Used IP blocks,

|  |  |
| --- | --- |
| IP BLOCK | Usage |
| clk | clk / reset signals for the SoC |
| timer0 | For interval timer operation |
| high\_res\_timer | To calculate time intervals between implementations (software/hardware) |
| sysid | Gives a unique ID to the SoC (prevents accidental downloading of software) |
| cpu | Processor for the SoC |
| jtag\_uart | UART connection with the computer |
| onchip\_mem | Data/instruction code save |

**Table01: Used IP Blocks**

**Hardware**

After creating the base system using above components on Qsys we have to implement a new component to handle the custom instruction.

The component is built using the provided Verilog files as **CRC\_Custom\_Instruction.v** & **CRC\_Component.v.** Using the files we can create a single interface with the name

**Nios\_custom\_instruction\_slave**

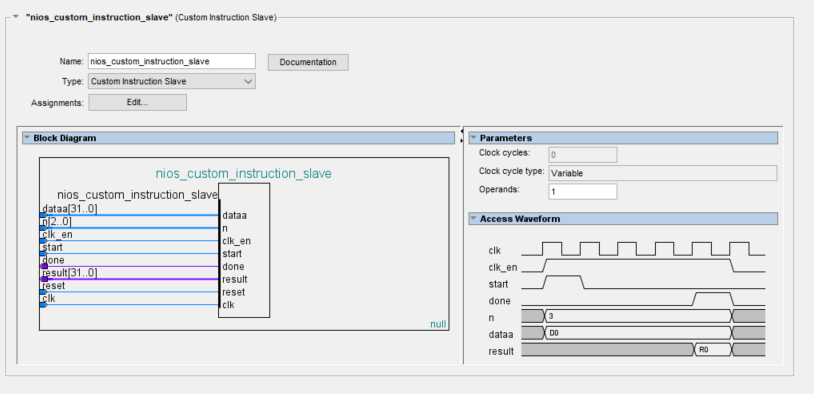
Of the type Custom Instruction Slave

As parameter the clock cycles are set to 0 and the clock cycle type set to Variable with operands as 1.

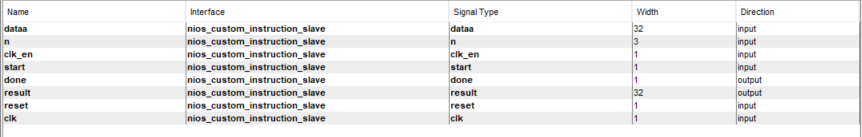
After defining the interface the signals were set with the interface signal types.

After the creating the component we can make the connection between its Custom instruction slave to CPU’s custom instruction master.

Then the BDF file can be completed by importing the newly generated SoC.



**Fig01: CRC Custom instruction Interface**



**Fig02: Signals of CRC Custom Instruction**

**Software**

For the software part after initializing the project provided files have to be imported.

|  |  |
| --- | --- |
| File | Description |
| ci\_crc.h | Header file for ci\_crc.c |
| ci\_crc.c | Access the CRC custom instruction |
| crc.h | Header for crc.c |
| crc.c | Software CRC algorithm run by the Nios II processor. |
| crc\_main.c | Main program that populates random test data and execute CRC in both software and custom instruction ,validates output and reports processing time. |

**Table02: Imported Files**

* Changes to be done would be setting the **high\_res\_timer** as the **timestamptimer** and **timer0** as **sys\_timer**.
* Once the BSP is generated we can build the project.
* Here we get our first error signifying undefined CRC\_MACRO(n,A)
* This is the function called to handle the CRC custom instruction on the hardware level.
* By looking through the error we can see that the definition of this function must be changed to the one provided in the system.h .
* The system.h file is generated along with the SoC when using Qsys.
* Going through the system.h file we get to see the commented CRC custom instruction part and the definition for the ALT\_CI\_CRC\_CUSTOM(n,A)
* Since this is what needs to be accessed using the the CRC\_MACRO(n,A) we can copy ALT\_CI\_CRC\_CUSTOM(n,A) and paste it at the definition in the ci\_crc.c

**Additional errors**

1. Errors when building found in BSP ->

These errors occurred because of redefining already defined variables that are already in the bsp. Since the bsp(board support package) always tries to optimize the software bsp gives out errors. Removing the redefinitions and cleaning and building the bsp solves this.

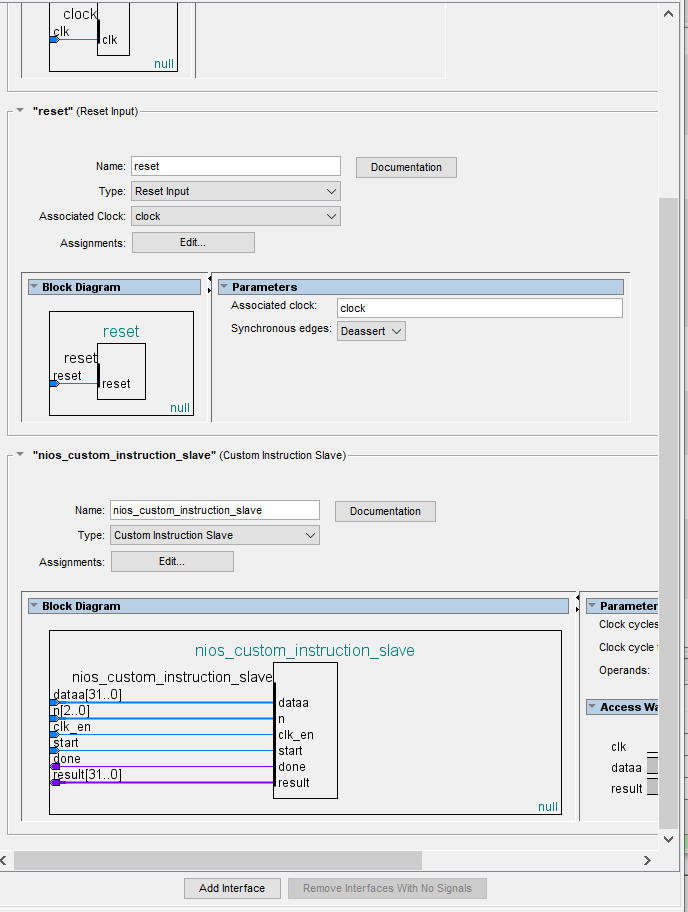
1. Errors on memory allocation ->

My first build I had used the default settings for onchip mem. This memory is not enough. Allocating a memory of 256KB solves this issue.

1. Output Error when run ->

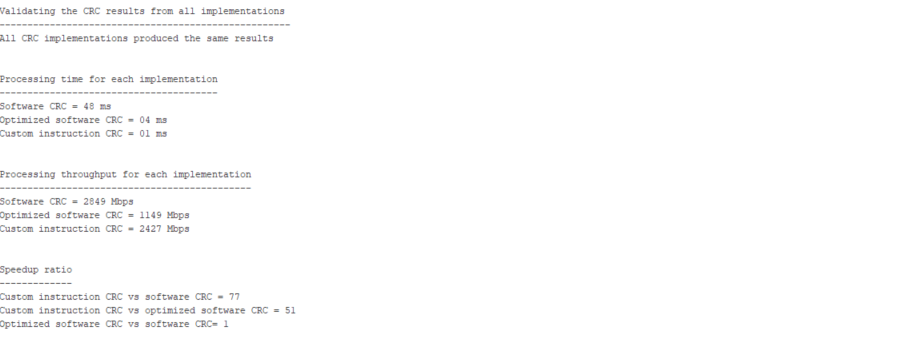
After resolving above errors and running the code on hardware the run report gave out an error “CRC outputs not same”.Going through the message shows that CRC custom instruction outputs -> 0x0

While software implementation gives different output. Going back to QSYS recognized that my implementation of custom instruction has three interfaces, CLK RESET and the custom instruction slave while the manual has these in one. Looking at the signals the CLK and RESET receive the signals separately rather than from the nios\_custom\_instruction\_slave. Removing these extra interfaces by assigning the correct signals corrected the error.



**Fig03: 3 Interface**

**Observations**

****

**Fig04: NiosII Console output**

The output reports the system specification

Clock speed = 50Hz

Number of buffer locations = 12

Size of each byte buffer = 256Bytes

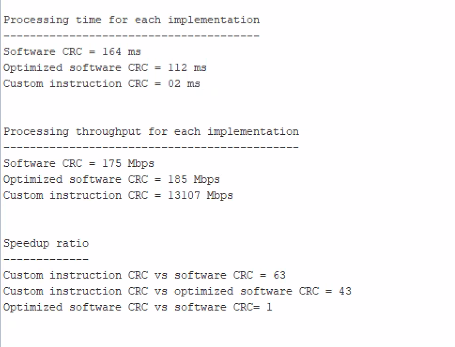
Shows the validation of each output of each method is the same.

**Processing time for each implementation**

Shows that the unoptimized software CRC is much slower than the optimized software CRC which is expected, as the optimized implementation uses a lookup table to access already calculated middle values. The custom instruction seems to be faster than the optimized implementation with a processing time of around 1 ms , which is a significant improvement.

**Processing throughput for each implementation**

The software implementation has a highest throughput followed by the custom instruction and with the least throughput optimized software implementation. Usually, this metric shows the throughput during a calculation. My implementation of the CRC algorithm shows that the software throughput has become higher than the custom instruction. Usually this is the other way around. Since the speedup ratios are same as expected results we can assume the above results are a result of some deviation in the acquired data paths.



**Fig05: Implementation results using SDRAM memory**

By looking at an implementation done using the SDRAM rather than the onchip memory as I have we see how different the results are when it comes to speed of the calculation and Procssing throughput. Here since we are accessing memory outside the SOC there is a significant calculation delay in reaching the software implemenataion.